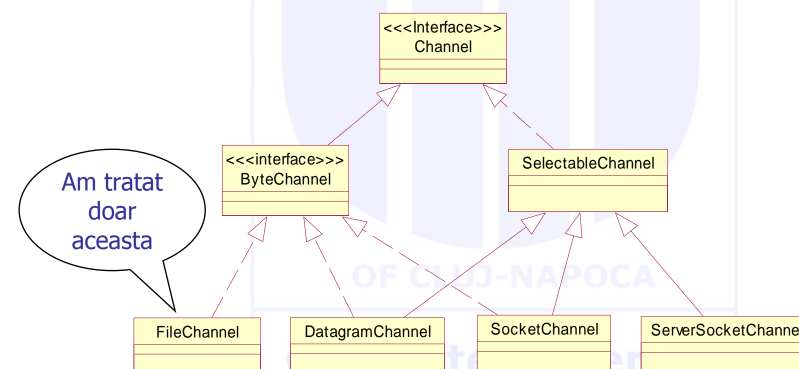
Programare orientată pe obiecte

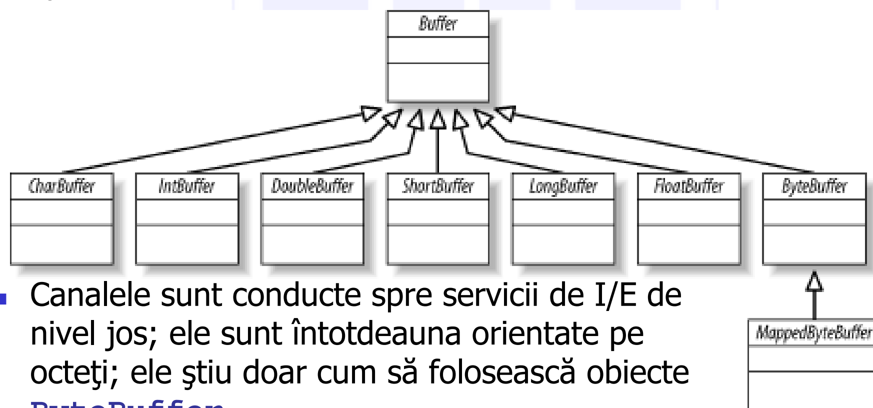
1. Mai multe despre noua I/E Java
2. Fire de lucru Java

Ierarhie simplificată pentru `Channel`



`Buffer`

- `Buffer` (zone tampon) au fost create în primul rând pe post de containere pentru datele trimise/recepţionate pe/de pe canale.



- Canalele sunt conducte spre servicii de I/E de nivel jos; ele sunt întotdeauna orientate pe octeţi; ele ştiu doar cum să folosească obiecte `ByteBuffer`.

Vederi pentru `Buffer`

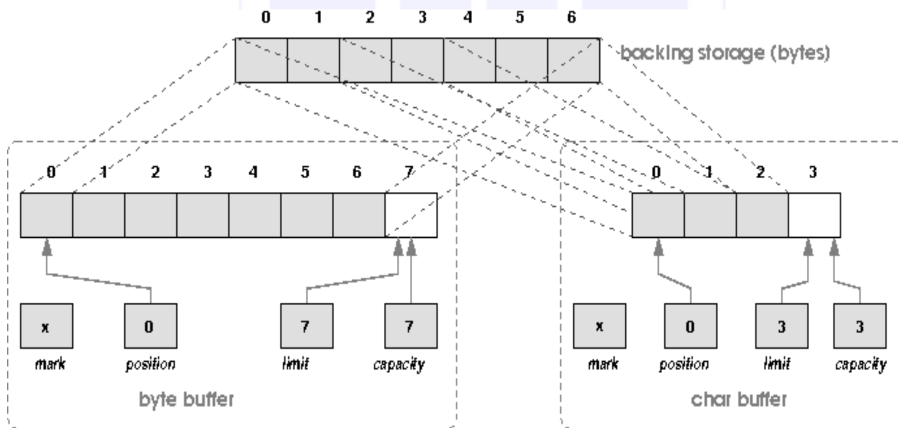
- Presupunem că avem un fişier care conţine caractere Unicode stocate ca valori pe 16 biţi (codificare UTF-16 nu UTF-8)
  - UTF = Unicode Transformation Format
- Pentru a citi o bucată din acest fişier în zona tampon putem crea o vedere `CharBuffer` a octeţilor respectivi:

```
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

  - Creează o vedere a `ByteBuffer` care se comportă ca un `CharBuffer` (combină fiecare pereche de octeţi din tampon într-o valoare caracter pe 16 biţi)
- Clasa `ByteBuffer` are metode de acces ad hoc la valorile primitive. D.e., pentru a accesa ca întreg patru octeţi dintr-o zonă tampon:

```
int fileSize = byteBuffer.getInt();
```

- Vedere **`CharBuffer`** a unui **`ByteBuffer`**

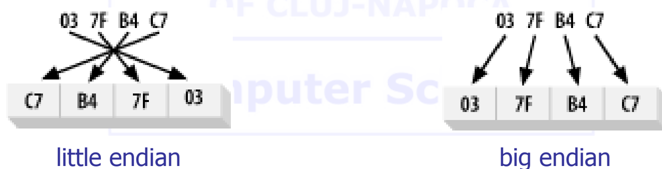


Exemplu de vederi pentru `Buffer`

```
import java.nio.*;
public class Buffers {
  public static void main(String[] args)  {
    try {
      float[] floats = {
        6.612297E-39F, 9.918385E-39F,
        1.1093785E-38F, 1.092858E-38F,
        1.0469398E-38F, 9.183596E-39F
      }
      ByteBuffer bb =
        ByteBuffer.allocate(floats.length * 4 );
      FloatBuffer fb = bb.asFloatBuffer();
      fb.put(floats);
      CharBuffer cb = bb.asCharBuffer();
      System.out.println(cb.toString());
    } catch (Exception e) {
      System.out.println(e.getMessage();
      e.printStackTrace();
    }
  }
}
```

Interschimbarea octeţilor

- *Endian-ness* : ordinea de combinare a octeţilor pentru a forma valori numerice mai mari
  - Când octetul cel mai semnificativ ca ordine numerică este primul stocat în memorie (la adresa mai mică) avem ordinea *big-endian*
  - Cazul opus, în care cel mai puţin semnificativ octet apare primul, este *little-endian*



little endian big endian

Vederi ale `Buffer` şi Endian-ness

- Fiecare obiect tampon are o setare a *ordinii octeţilor*.
  - Cu excepţia lui `ByteBuffer`, proprietatea poate fi numai citită şi nu se poate schimba.
- Setarea ordinii octeţilor la obiectele `ByteBuffer` poate fi modificată oricând.
  - Aceasta afectează ordinea rezultată pentru orice vederi create pentru acel obiect `ByteBuffer`.
  - Dacă datele Unicode din fişier au fost codificate ca UTF-16LE (little-endian), trebuie să setăm ordinea pentru `ByteBuffer` înainte de a crea vederea `CharBuffer`:

```
byteBuffer.order (ByteOrder.LITTLE_ENDIAN);
CharBuffer charBuffer = byteBuffer.asCharBuffer();
```

- Noua vedere moşteneşte ordinea lui `ByteBuffer`
- Setarea ordinii octeţilor la momentul apelului *afectează* modul de combinare pentru formarea valorii returnate sau divizate pentru ceea ce este stocat în zona tampon.

Zone tampon directe

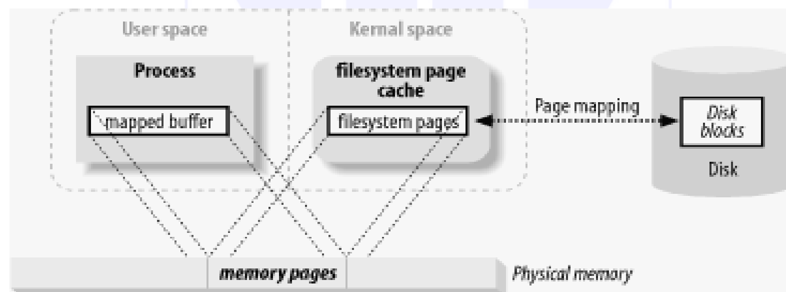
- Datele încapsulate de o zonă tampon pot fi stocate în câteva moduri:
  - într-un tablou privat creat de obiectul `Buffer` (alocare),
  - într-un tablou furnizat de programator (împachetare), sau,
  - în cazul zonelor tampon *directe*, în *spaţiu de memorie nativă* din afara JVM.
- La crearea unei zone tampon directe (prin apelul `ByteBuffer.allocateDirect()`), se alocă memorie sistem nativă şi se împachetează într-o zonă tampon.
- Scopul principal al zonelor tampon directe este efectuarea de I/O peste canale.
- Implementările canalelor pot face ca operaţiile de I/E de la nivelul SO să acţioneze direct asupra spaţiului de memorie nativă al unei zone tampon directe.

Fişiere mapate pe memorie

- `MappedByteBuffer` este o formă specializată de `ByteBuffer`.
  - În majoritatea sistemelor de operare se poate *mapa pe memorie* un fişier folosind un *apel sistem* de genul lui `mmap()` (sau ceva asemănător – atenţie, acest apel nu este Java) pe un descriptor de fişier deschis.
    - Apelul lui mmap() returnează un pointer la un segment de memorie care reprezintă de fapt conţinutul fişierului.
  - Extragerile din locaţiile de memorie din zona respectivă vor întoarce datele din fişier de la deplasamentul corespunzător.
  - Modificările efectuate asupra spaţiului de memorie sunt scrise în fişierul de pe disc.

Fişiere mapate pe memorie

- Alte procese care rulează în spaţiul utilizator vor mapa pe acelaşi spaţiu de memorie fizică, prin acelaşi cache al sistemului de fişiere şi, de aceea pe aceleaşi date de pe disc.
- Fiecare dintre procese va vedea modificările efectuate de oricare alt proces.
- Aceasta poate fi exploatata ca o formă de memorie partajată persistentă.



Exemplu `MappedByteBuffer`: inversarea octeţilor dintr-un fişier

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class ReverseBytes {
   public static void main(String
     args[]) throws IOException {
      // verifica argumentul din
     // linia de comanda
      if (args.length != 1) {
         System.err.println(
            "Lipseste numele fisierului");
         System.exit(1);
      }
      // get channel
      RandomAccessFile raf =
           new RandomAccessFile(
                   args[0], "rw");
      FileChannel fc = raf.getChannel();
      // mapeaza fisierul la buffer
      MappedByteBuffer mbb =
      fc.map(FileChannel.MapMode.READ_W
      RITE, 0,  fc.size());
      // inverseaza octetii din fisier
      int len = (int)fc.size();
      for ( int i = 0, j = len - 1;
                 i < j;
                 i++, j--)
      {
         byte b = mbb.get(i);
         mbb.put(i, mbb.get(j));
         mbb.put(j, b);
      }
      // finish up
      fc.close();
      raf.close();
   }
}
```

Citiri distribuitoare (scatering) şi scrieri colectoare (gathering)

- D.e., cu o singură cerere de citire de pe canal putem pune primii 32 octeţi în tamponul `header`, următorii 768 octeţi în tamponul `colorMap` şi restul în `imageBody`
- Canalul umple fiecare tampon pe rând până când toate sunt pline sau nu mai sunt date de citit

```
. . .
ByteBuffer header = ByteBuffer.allocate (32);
ByteBuffer colorMap = ByteBuffer (256 * 3);
ByteBuffer imageBody = ByteBuffer (640 * 480);
ByteBuffer [] scatterBuffers = { header, colorMap,
  imageBody };
fileChannel.read (scatterBuffers);
```

Transferuri directe pe canale

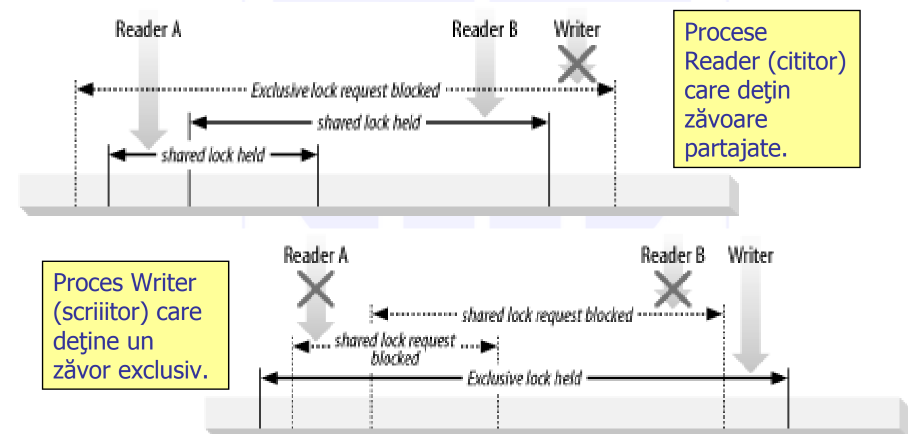
- Transferul pe canal permite interconectarea a două canale astfel încât datele să fie transferate direct dintr-un canal în celălalt fără intervenţii suplimentare.
- Deoarece metodele `transferTo()` şi `transferFrom()` apartin clasei `FileChannel`, trebuie ca sursa şi destinaţia unui transfer pe canal să fie obiecte `FileChannel` (d.e., nu se poate transfera de la un socket la altul).
- Celălalt capăt poate fi orice `ReadableByteChannel` sau `WritableByteChannel`, după nevoi.
- Notă: demo în subdirectorul nio

Exemplu de transfer direct între canale

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
public class DirectChannelTransfer {
 public static void main(String args[])  throws IOException {
   // verifica argumentele de pe linia de comanda
   if (args.length != 2) {
     System.err.println("Lipsesc numele de fisiere");
     System.exit(1);
   }
   // get channels
   FileInputStream fis =  new FileInputStream(args[0]);
   FileOutputStream fos =  new FileOutputStream(args[1]);
   FileChannel fcin = fis.getChannel();
   FileChannel fcout = fos.getChannel();
   // executa copierea fisierului
    fcin.transferTo(0, fcin.size(), fcout);
   // incheie
   fcin.close();
   fcout.close();
   fis.close();
   fos.close();
 }
}
```

Metoda **transferTo** transferă octeţi din canalul sursă (**fcin**) în canalul destinaţie specificat (**fcout**). Transferul este executat tipic *fără citiri şi scrieri explicite la nivel utilizator* pe canal.

Zăvorârea fişierelor



- Zăvoarele pentru fişiere (file locks) sunt necesare în general la integrarea cu aplicaţiile non-Java, pentru a media accesul la fişiere de date partajate.

Expresii regulate

- Expresiile regulate (`java.util.regex`) sunt parte a NIO
- Clasa `String` ştie de expresii regulate prin adăugarea următoarelor metode:

```
package java.lang;
public final class String implements java.io.Serializable,
  Comparable, CharSequence
{
  // Lista partiala din API
  public boolean matches (String regex)
  public String [] split (String regex)
  public String [] split (String regex, int limit)
  public String replaceFirst (String regex, String
  replacement)
  public String replaceAll (String regex, String
  replacement)
}
```

Exemple de expresii regulate

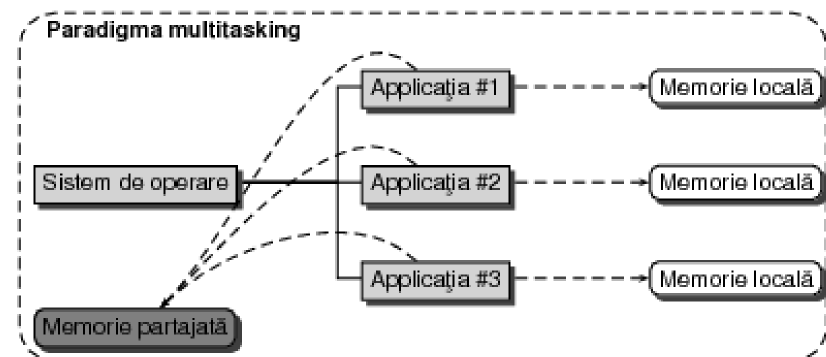
```
public static final String VALID_EMAIL_PATTERN = "([a-zA-Z0-9_\\-
   \\.]+)@((\\[[0-9]{1,3}\\.[0-9]" + "{1,3}\\.[0-9]{1,3}\\.)|(([a-zA-
   Z0-9\\-]+\\.)+))" + "([a-zA-Z]{2,4}|[0-9]{1,3})(\\]?)";
...
if (emailAddress.matches (VALID_EMAIL_PATTERN))
{
  addEmailAddress (emailAddress);
}
else
{
  throw new IllegalArgumentException (emailAddress);
}

// imparte sirul lineBuffer (care contine o serie de valori separate prin
   virgule) in subsiruri si returneaza sirurile respective intr-un tablou
String [] tokens = lineBuffer.split ("\\s*,\\s*");
```

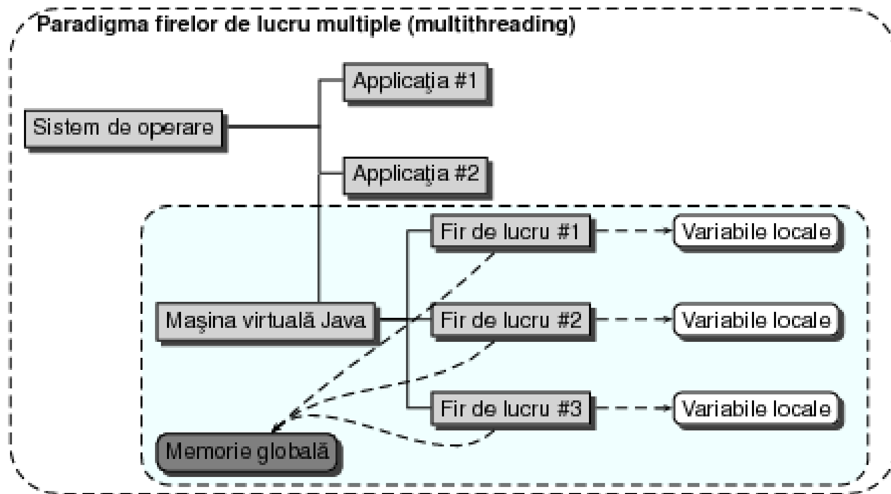
Introducere în firele de lucru Java

Multitasking

- Mai multe procese par a se executa simultan, la nivelul sistemului de operare



Multitasking vs Multithreading



Sarcini ale firelor de lucru

- Firele de lucru sunt utile în mai multe feluri:
  - Acolo unde trebuie să se întâmple mai multe lucruri simultan.
    - D.e., o aplicaţie multimedia poate necesita ca procesele audio, video şi de control să se execute în paralel. Există adesea perioade de aşteptare a răspunsului sistemelor de IE mai lente, timp în care procesorul poate face altceva.
  - Programe cum sunt sistemele server/client sunt mult mai uşor de proiectat şi scris folosind fire de lucru.
  - Algoritmi matematici, cum sunt sortarea, căutarea numerelor prime etc. sunt potrivite pentru prelucrarea paralelă.
  - Pe sistemele multi-procesor, maşinile virtuale Java pot rula firele de lucru pe procesoare diferite şi pot obţine astfel prelucrare paralelă adevărată şi creşteri de performanţe semnificative faţă de platformele mono-procesor.

Multithreading în Java

- Proprietăţile firelor de lucru multiple din programele Java:
  - Fiecare fir de lucru îşi începe execuţia la o locaţie bine cunoscută, predefinită
  - Fiecare fir de lucru îşi execută codul începând de la locaţia de start, într-o secvenţă ordonată, predefinită (pentru un set de date de intrare dat)
  - Fiecare fir de lucru îşi execută codul independent de celelalte fire de lucru din program
  - Firele de lucru par a avea un anumit grad de simultaneitate în execuţie
  - Firele de lucru au acces la diferite tipuri de date

Multithreading în Java

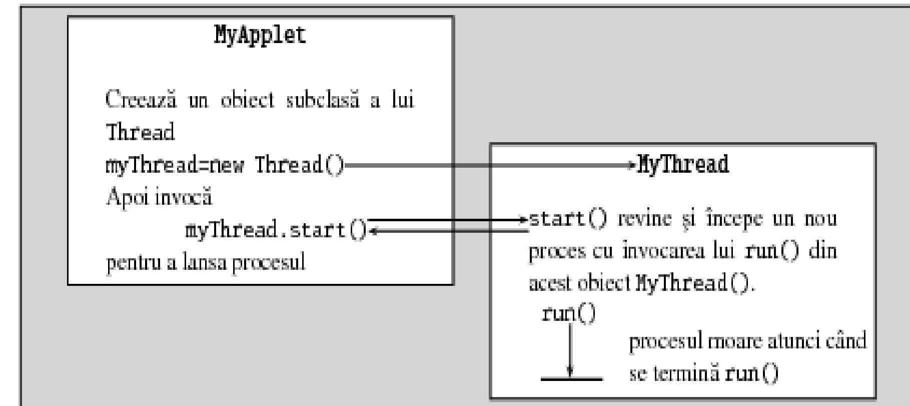
- Toate programele Java în afara aplicaţiilor simple cu intrare-ieşire pe consolă sunt aplicaţii multithreading.
- Procesele *grele* (*heavyweight*) se rulează direct sub sistemul de operare al maşinii locale.
- Fir de lucru: un singur curs secvenţial al controlului, numit şi proces *uşor* (*lightweight*), lansat din procesul principal
  - Sunt procese paralele care rulează *înăuntrul* unui program
  - În Java se pot crea fire de lucru din program, aşa cum se pot rula programe sub un sistem de operare
- Java: creează fire de lucru în două moduri:
  - Clasa extinde clasa `Thread` şi îi suprascrie metoda `run()`.
  - Clasa implementează interfaţa `Runnable`, care are o singură metodă: `run()`.
    - **Clasa îi transmite o referinţă la sine însuşi când creează un fir de lucru.**
    - **Apoi firul apelează înapoi metoda `run()` din clasă.**

Sub-clasarea clasei Thread

- Metoda `run()` din fir corespunde metodei `main()` pentru o aplicaţie.
- La pornirea firului, acesta invocă metoda `run()`, iar atunci când procesul revine din metoda `run()` firul de lucru *moare*.
  - Nu se poate "învia" un fir de lucru mort. În loc de "înviere", trebuie creată o nouă instanţă de fir de lucru.
- Subclasa trebuie să suprascrie metoda `run()`.
- În metoda `run()`se pune codul care trebuie executat în paralel cu programul principal

Exemplu de subclasare a clasei Thread

- Demo: subclass

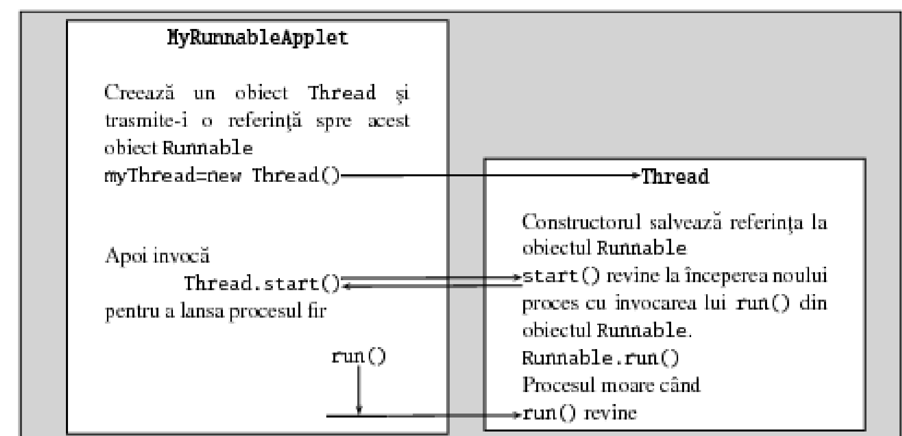


Implementarea interfeţei Runnable

- Implementăm interfaţa `Runnable` şi îi suprascriem metoda `run()`
- Transmitem o referinţă la o instanţă a respectivei implementări a `Runnable` spre o instanţă de fir de lucru şi firul de lucru *apelează înapoi (calls back)* metoda `run()` din obiectul `Runnable`.
- Firul de lucru moare, ca la procedeul anterior, la revenirea procesului din `run()`.
- Procedeul este convenabil în cazurile în care dorim să creăm un singur tip de fir de lucru, cum este, spre exemplu, *animaţia dintr-un applet*.

Exemplu de implementare a interfeţei Runnable

- Demo: runnable



Interfaţa `Runnable`: Schiţă de implementare

```
public class ClassToRun extends SomeClass implements
  Runnable
{ . . .
  public void run()
  {
    // Completati aici dacă ClassToRun
    // a fost derivată din Thread
  }
  . . .
  public void startThread()
  {
    Thread theThread = new Thread(this);
    theThread.run();
  }
  . . .
}
```

Sub-clasarea vs. `Runnable`

- Tehnica folosind `Runnable` este în special convenabilă dacă dorim să creăm un singur fir care să efectueze o anumită sarcină de lucru.
  - Metoda `run()` va avea acces la variabilele instanţă din obiectul `Runnable`.
  - D.e., pentru animaţie în applet, faceţi applet-ul `Runnable`.
    - **Metoda `run()` are atunci acces la variabilele applet-ului, care pot fi parametri transmişi prin etichetele applet-ului sau stabiliţi de utilizator prin intermediul interfeţei grafice**
- Dacă doriţi sa creaţi mai multe fire de lucru, atunci de obicei e mai bine să derivaţi din clasa `Thread`.
  - Aceasta ajută la conceptualizarea mai bună a firelor ca obiecte independente
  - Puteţi seta valorile oricăror parametri de care ar fi nevoie folosind constructori sau metode "setter" (de setare)

Stoparea/Punerea în pauză a unui fir de lucru

- Un fir se opreşte în trei moduri:
  - Dacă revine normal din `run()`.[Cea mai bună cale]
  - Se apelează metoda `stop()` a firului. (Acum depreciată (deprecated). Nu o folosiţi.)
  - Dacă este întrerupt de o excepţie neinterceptată.
- Dacă metoda `run()` conţine o buclă de durată sau una infinită – opriţi buclarea atunci când se modifică o variabilă care poate fi modificată de procesul principal. D.e. o valoare `boolean` setată la `false` sau o variabilă referinţă setată la `null` pentru oprire
- Opriţi explicit întotdeauna firele din applet-uri la apelul metodei `stop()` a *applet-ului.*
  - În caz contrar, firele pot continua să ruleze chiar dacă browserul încarcă o noua pagină de Web

Stoparea/Punerea în pauză a unui fir de lucru

- Pentru cazul `Runnable`, puteţi porni un nou fir cu setările variabilelor la valorile pe care le-au avut în momentul opririi firului precedent.
- În acest caz, oprirea unui fir şi pornirea unuia nou acţionează ca acţiuni *pauză/start*.
- Metodele clasei `Thread` *suspend()* şi *resume() au fost* depreciate (deprecated) pentru a evita problemele de interblocare a firelor de lucru

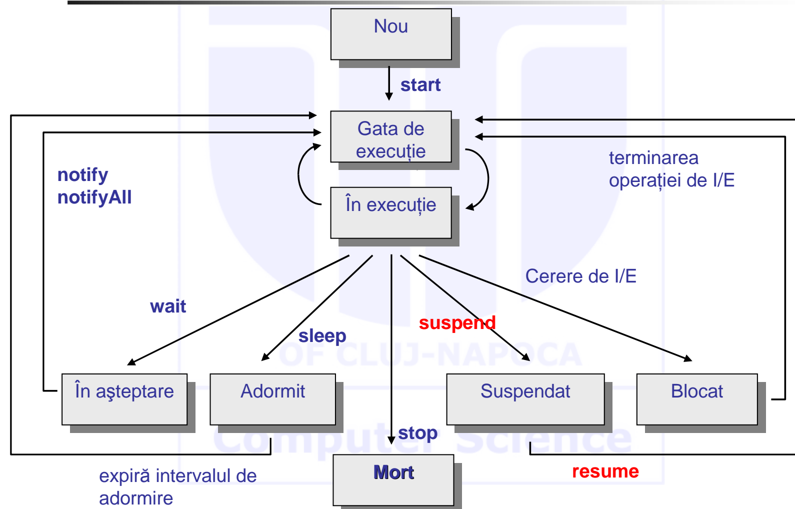
Thread.sleep

- `Thread.sleep` este o metodă statică din clasa `Thread` care pune în pauza un fir de lucru care conţine invocarea
  - Pune în pauză firul timp de numărul de milisecunde dat ca argument
  - Remarcaţi că metoda poate fi invocată dintr-un program obişnuit pentru a insera o pauză în singurul fir al programului respectiv
- Metoda poate arunca o excepţie verificată, `InterruptedException`, care trebuie declarată sau interceptată
  - Atât clasa `Thread` cât şi `InterruptedException` se află în pachetul `java.lang`

Stările firelor de lucru

- `getState():` (Java 5) metoda întoarce un `Enum` de `Thread.States`. Stări:
  - NEW – fir "proaspăt" care nu şi-a început încă execuţia.
  - RUNNABLE – fir în curs de execuţie în JVM.
  - BLOCKED – fir blocat care aşteaptă după un zăvor monitor.
  - WAITING – fir care aşteaptă să fie notificat de un alt fir.
  - TIMED_WAITING – fir care aşteaptă un anumit timp să fie notificat de un alt fir.
  - TERMINATED – fir a cărui metodă run s-a terminat.

Ciclul de viaţă al unui fir de lucru



Planificatorul de fire

- Planificatorul de fire execută fiecare fir de lucru pentru un interval de timp scurt (o *felie de timp (time slice*))
- Apoi planificatorul activează un alt fir
- Întotdeauna vor fi variaţii ale timpilor de rulare, în special la apelul serviciilor sistemului de operare (d.e. pentru operaţii de intrare şi de ieşire)
- Nu sunt garanţii referitoare la ordinea de execuţie a firelor

Terminarea firelor de lucru

- Un fir de lucru se termină la terminarea metodei sale `run()`
- *Nu terminaţi* un fir de lucru folosind metoda depreciată `stop`
- În loc de aceasta, notificaţi firul de lucru că ar trebui să se termine folosind

```
t.interrupt();
```

- `interrupt()` nu face ca firul să se termine – metoda setează un câmp boolean în structura de date a firului

Terminarea firelor de lucru

- Metoda `run()` ar trebui să verifice ocazional dacă firul de lucru a fost întrerupt
  - Folosiţi metoda `interrupted()`
  - Un fir de lucru care a fost întrerupt ar trebui să elibereze resursele pe care le foloseşte, să "cureţe" şi să se termine

```
public void run()
{
   for (int i = 1;
         i <= REPETITIONS && !Thread.interrupted(); i++)
   {
      Do work
   }
   Clean up
}
```

Terminarea firelor de lucru

- Metoda `sleep` aruncă o excepţie de tipul `InterruptedException` atunci când un fir "adormit" este întrerupt
  - Interceptează excepţia
  - Termină firul de lucru

```
public void run() {
   try {
      for (int i = 1; i <= REPETITIONS; i++)
      {
         Do work
      }
   }
   catch (InterruptedException exception) {
   }
   Clean up
}
```

Terminarea firelor de lucru

- Java nu forţează terminarea unui fir atunci când acesta este întrerupt
- Este treaba firului de lucru ce anume face atunci când este întrerupt
- Întreruperea reprezintă un mecanism general pentru a obţine "atenţia" firului de lucru întrerupt

Animaţii

- Animaţiile sunt o sarcină uzuală pentru firele de lucru: ele sunt folosite la controlul animaţiei
  - Un fir de lucru poate dirija desenarea fiecărui cadru în timp ce alte aspecte, cum sunt
  - Răspunsul la interacţiunea cu utilizatorul poate continua în paralel
- Demo: clock, drop2d, sunsort

Rezumat

- Mai mult despre noua IE Java
  - Buffer (zonă tampon)
    - Vederi
    - Endian-ness
    - Direct ~
    - Memory mapped ~
  - Canale la fişiere
    - Fişiere mapate pe memorie
    - Transferuri directe
    - Zăvorârea
  - Clasa `String` – expresii regulate
- Introducere la firele de lucru Java
  - Multitasking vs. multithreading
  - Crearea firelor de lucru:
    - Sub-clasarea `Thread`
    - Implementarea `Runnable`
  - Stoparea/punerea în pauză a firelor
  - Terminarea firelor
  - Animaţie simplă cu fire de lucru